V.10.

Общий обзор занятия.

Раньше приблизительно в этом месте мы разбирались с фракталами. Рисовали звёздочку Коха, кривую Леви, кривую дракона, коврик и салфетку Серпинского. Все они самоподобны, по крайней мере теоретически, да и практически они самоподобны, по крайней мере на глаз. И, благодаря этому, все они являются классическими примерами рекурсии, которая просто лежит в основе из построения. Но появились у меня, увы, сомнения. И сомнения эти технического плана. На 5-ом семестре довольно часто есть люди, которые ещё не были на 4-ом. И если в Паскале технические сложности запуска графики были минимальными, то в Go, точнее в package sdl, они гораздо более громоздкие. Да, конечно, можно совершенно формально забросить всю техническую часть нициализации и разрушения графического окна, но там довольно здоровый кусок, который придётся тупо копипастить, а это не очень, но всё-таки неприятно. Но тут ещё один момент. Давно лежит в запасе здоровенный сюжет, связанный с *Ханойскими башнями* и подобными им задачами. И рекурсия там совершенно естественная и интересная, и задач там много, в общем, сюжет богатый. И решил я воспользоваться моментом и поставить в это место как раз разговоры (и действия) вокруг *Ханойских башен*. Заодно и зависимость 5-го семестра от 4-го существенно снижается. Тем не менее, совсем фракталы выкидывать тоже жалко. Поэтому я ставлю в материалы к этому занятию небольшой план, что и как можно делать с фракталами (там много картинок, поэтому я кладу этот текст в docx и в pdf форматах), привожу пример программы, строящей звёздочку Коха и выкладываю эти материалы в отдельном каталоге Fractals. Кто хочет - может их

использовать, там довольно кратко, но все основные моменты есть.

А основная линия занятия формулируется легко. Рассматриваем классическую задачу о Ханойских башнях, находим там рекурсию. Кстати, приятный момент - она там не лежит явно в описании задачи, но как только мы задумаемся над алгоритмом решения, мы приходим к рекурсии практически сразу. Приятно - нужна некая интеллектуальная работа, но она срабатывает быстро, эффективно и довольно неожиданно, что особенно обостряет интерес. И как раз там очень чётко, явно и эффективно работает как раз “думание с конца”, движенит требуемого результата к организации всего процесса, столь характерное дл рекурсии. Именно от конца - от результата, а не от начала - от формулировки задачи, как, скажем в часто применяемых примерах типа чисел Фибоначии или вычисления факториала. IMHO, это не есть удачные в методическом плне примеры для начала разборок с рекурсией, хотя позже, в разговорах об опасностях рекурсии, числа Фибоначчи мы применим. Строим решение классической задачи, а потом спокойно по проторенной дорожке разбираем всевозможные вариации на тему классического сюжета. Вариаций много - можно спокойно выбрать по вкусу… Все вариации разобрать за время лекции невозможно и не нужно. Какие-то варианты оставляем в качестве упражнений. Конечно, совершенно необходимо разобрать спокойно классическую задачу. Далее идут три довольно однотипных задачи - два “Ремонта в Ханое” и циклическая задача. Вот их все три точно не стОит разбирать - одну, максимум две, вполне достаточно. Сортирующие башни - вот это интересный вариант, его стоит разобрать. А вариант этой задачи (в тексте я приведу только формулировку) - уместен для самостоятельной работы. С несправедливыми башнями непонятно - этот вариант во многом продолжает серию задач из двух ремонтов и циклических башен, так что можно разбирать, можно не разбирать,

кажется, времени и сил уже не хватает, но мало ли… А вот обменные башни рассмотреть есть смысл: и сюжет несколько отличается, и решение получается просто и естественно, так что хотелось бы его обсудить хотя бы вкратце, впрочем, подробно тут и не требуется.

А ещё я привожу несколько задач-головоломок более или менее продолжающих линию Ханойских башен. Две из них - “Светофор” и “Переправа” - совершенно изоморфны Ханойски башням, но это, во-первых, надо увидеть, и об этом есть смысл поговорить, а, во-вторых, в них требуется изменить формат вывода, что требует довольно чёткого понимания происходящего в рекурсивной программе, и это хорошо. Решения других головоломок находятся вполне в русле идей решения Ханойских башен, так что эти задачи хороши джля самостоятельной работы - в них ребуется только слегка обсудить алгоритм решения, явно, может быть даже акцентированно, придерживаясь намеченнойлинии размышлений.

Повторю, однако, вполне приемлемо в Ханойские башни сильно не влезать (да хоть и вовсе не влезать, хотя их жалко), а провести занятие, базируясь на фрактальной теме. Но Ханойских башен жалко…

Лекция.

Ханойские башни. Классическая задача.

Постановка задачи.

В классической формулировке задача выглядит следующим образом:

- Даны три стержня, на один из которых нанизаны восемь колец, причём кольца отличаются размером и лежат меньшее на большем.
- Один ход состоит в том, чтобы перености ровно одно кольцо. При этом нельзя класть большее кольцо на меньшее.
- Задача состоит в том, чтобы перенести всю пирамиду на другой стержень. Желательно за наименьшее количество ходов.

Мы будем рассматривать чуть более общую задачу с N кольцами вместо восьми.

Ссылка на Википедию. Там и картинка красивая есть..
На всякий случай вот текст ссылки: `https://ru.wikipedia.org/wiki/Ханойская_башня` .
А вот - картинка из Вики в отдельном файле

Классическая задача. Решение.

Итак, мы хотим переложить всю пирамиду с первого стержня на третий. В какой-то момент мы должны будем переносить самое большое кольцо. Чтобы перенести самое большое кольцо, необходимо, чтобы это кольцо было верхним на своём стержне. Поскольку ниже самого большого кольца не может лежать ничего, то это кольцо лежит на своём стержне одиноко. Переложить его можно только на пустой стержень - мы же не можем класть самое большое кольцо ни на какое другое кольцо. Значит все остальные кольца лежат на оставшемся диске. А лежать они могут только в порядке убывания снизу вверх. Это означает, что для того, чтобы иметь возможность перенести само большое кольцо

с первого стержня на третий, необходимо предварительно перенести башню из верхних N-1 колец с первого стержня на второй. И тогда, после переноса самого большого кольца на третий стержень, остаётся переместить башню из N-1 колец со втрого стержняя а третий. Самое большое кольцо никак этим процессам не мешает - на то оно и самое большое, на него можно класть любые кольца.

И сразу получаем таким образом алгоритм решения:

для того, чтобы **переложить пирамиду из N колец с первого стержня на третий** необходимо:

1. переложить все, что выше самого большого диска (а это - пирамида высоты N-1) с первого стержня на второй, используя третий стержень как вспомогательный
2. переложить самый большой диск с первого стержня на третий
3. переложить пирамиду высоты N-1 со второго стержня на третий, пользуясь первым стержнем, как вспомогательным.

Заметим, что все эти шаги не просто решают задачу, но и совершенно необходимы.
Замечание. Формально говоря, не совсем необходимы. Можем, например, решать так: перемещаем башню высоты N-1 с первого стержня на третий, переносим кольцо N на второй стержень, перемещаем башню высоты N-1 с третьего стержня на первый, переносим кольцо N на трнтий стержень, перемещаем башню высоты N-1 на третий стержень. Но этот алгоритм очевидно требует больше операций.

Иллюстрация решения - здесь

Код здесь: **пример 01.classic.go**

```
package main

import (
    "fmt"
)

func Hanoy(n int, start, transit, finish int) {
    if n > 0 {
        Hanoy(n-1, start, finish, transit)
        fmt.Println(start, finish)
        Hanoy(n-1, transit, start, finish)
    }
}

func main() {
    Hanoy(8, 1, 2, 3)
}
```

Ханойские башни. Вариации на тему.

Ханойские башни. Ремонт в Ханое - 1.

Запрещено перекладывать кольца с первого стержня сразу на третий.

Решение.

В принципе подход к решению уже есть - перемещать меньшие башни рекурсивно, чтобы дать возможность перекладывать бОльшие кольца - делать настоящие (нерекурсивные) ходы. Опять рассмотрим перемещения самого большого кольца. С первого стержня на третий его переносить нельзя. А в нашем решении классической задачи

присутствует ход со стержня start на стержень finish. Значит, если start = 1, а finish = 3, то надо придумывать что-то другое. А вот во всех остальных случаях алгоритм решения классической задачи прекрасно работает. Что же делать, если start = 1, а finish = 3? Самое большое кольцо надо переносить с первого стержня на второй, а потом на третий. Значит

- сначала надо рекурсивно переложить пирамиду высоты N-1 с первого стержня на третий,
- затем таки перенести самое большое кольцо с первого стержня на второй,
- потом переложить пирамиду высоты N-1 обратно с третьего стержня на первый,
- далее перенести самое большое кольцо со второго стержня на третий,
- и, наконец, опять рекурсивно переложить пирамиду высоты N-1 с первого стержня на третий.

Вот вам и алгоритм. Выписывать его смысла нет - проще посмотреть код.

Иллюстрация решения - здесь

Код здесь: **пример 02.repair1.go**

```
package main

import (
    "fmt"
)

func Hanoy(n int, start, transit, finish int) {
    if n > 0 {
        if start == 1 && finish == 3{
```

```
            Hanoy(n-1, 1, 2, 3)
            fmt.Println("1 2")
            Hanoy(n-1, 3, 2, 1)
            fmt.Println("2 3")
            Hanoy(n-1, 1, 2, 3)
        } else {
            Hanoy(n-1, start, finish, transit)
            fmt.Println(start, finish)
            Hanoy(n-1, transit, start, finish)
        }
    }
}

func main() {
    Hanoy(3, 1, 2, 3)
}
```

Ханойские башни. Ремонт в Ханое - 2.

Запрещено перекладывать кольца и с первого стержня сразу на третий, и обратно - с третьего на первый.

Решение.

И формулировка задачи похожа на предыдущую, и решение, соотвественно, аналогично. Просто к случаю start = 1, finish = 3 добавляется случай start = 3, finish = 1.
Иллюстрация решения - здесь
Вот код: **пример 03.repair2.go**

```
package main
```

```
import (
    "fmt"
)

func Hanoy(n int, start, transit, finish int) {
    if n > 0 {
        if (start == 1  && finish == 3) || (start == 3  && finish
== 1) {
        // variant: if transit == 2
            Hanoy(n-1, start, transit, finish)
            fmt.Println(start, transit)
            Hanoy(n-1, finish, transit, start)
            fmt.Println(transit, finish)
            Hanoy(n-1, start, transit, finish)
        } else {
            Hanoy(n-1, start, finish, transit)
            fmt.Println(start, finish)
            Hanoy(n-1, transit, start, finish)
        }
    }
}

func main() {
    Hanoy(3, 1, 2, 3)
}
```

Ханойские башни. Циклические башни.

Разрешены только перекладывания “по часовой стрелке”: с первого стержня на второй, со второго на третий или с третьего на первый.

Решение.

И опять практически то же самое: классический вариант работает, если start = 1, finish = 2, либо start = 2, finish = 3, либо start = 3, finish = 1. В остальных же случаях применяем “обходной” вариант.

Иллюстрация решения - здесь

Код здесь: **пример 04.cyclic.go**

```
package main

import (
    "fmt"
)

func Hanoy(n int, start, transit, finish int) {
    if n > 0 {
        if (start == 1  && finish == 2) || (start == 2  && finish
== 3) || (start == 3  && finish == 1) {
            Hanoy(n-1, start, finish, transit)
            fmt.Println(start, finish)
            Hanoy(n-1, transit, start, finish)
        } else {
            Hanoy(n-1, start, transit, finish)
            fmt.Println(start, transit)
            Hanoy(n-1, finish, transit, start)
            fmt.Println(transit, finish)
            Hanoy(n-1, start, transit, finish)
        }
    }
}

func main() {
    Hanoy(3, 1, 2, 3)
}
```

Ханойские башни. Сортирующие башни.

Слово “сортирующие” в данном случае не означает сортировки в порядке возрастания или убывания, как мы привыкли. А означает оно, что мы разделяем, распределяем кольца каким-то образом. В данном случае, чётные в итоге должны оказаться на стержне 2, не чётные - на стержне 3. Я бы назвал такие башни разделяющими или сепарирующими, но такова установившаяся терминология, так что будем пользоваться ею.

Формулировка задачи. Начальная позиция - классическая, все кольца находятся на стержне 1. Переложить кольца так, чтобы все чётные кольца оказались на стержне 2, а все нечётные - на стержне 3. Разумеется, большее кольцо не может лежать на меньшем.

Решение.

А вот тут всё несколько иначе. Но мы уже умеем решать подобные задачи. И в этом смысле задание важное, его хорошо бы включить в корпус лекции.

Что мы имеем в начале? Допустим, для определённости, что число N чётное. Тогда нам надо кольцо N переложить на старжень 2. Для этого надо переместить пирамиду из N-1 кольца на стержень 3. Но мы это умеем делать - обычным, классическим способом. Умеем - делаем. В итоге кольцо N лежит на стержне 2, а на стержне 3 находится башня из N-1 кольца. Остаётся переложить с третьего стержня пирамиду высоты N-1. Переложить сортирующим образом - чётные кольца на стержень 2, нечётные - на стержень 3.
Совершенно аналогично решается случай нечётного N: перемещаем башню из N-1 колец на стержень 2, перекладываем кольцо N на

стержень 3 и остётся переместить башню из N-1 колец со стержня 2 сортирующим образом.

Обратите внимание, на формулировки - в них чётко появилось то, что очевидно и так: процесс сортирующего перекладывания полностью определяется только стартовым стержнем - понятий финишный и промежуточный стержеь в этом задании вовсе нет. Чтобы не было сомнений, процитирую пару фраз из предыдущего абзаца: *переложить с третьего стержня пирамиду высоты N-1, переложить сортирующим образом*, *переместить башню из N-1 колец со стержня 2 сортирующим образом*.

А что делать, чтобы переложить башню высоты N сортирующим образом, если процесс стартует со стержня 2? Тоже понятно. Если N чётно, то нижнее кольцо оставляем на месте, а башню из N-1 колец переносим сортирующим образом. Если же N нечётно, то перемещаем классическим образом башню из N-1 колец на стержень 1, перекладываем кольцо N на третий стержень, а потом перемещаем башню высоты N-1 со стержня 1 сортирующим образом.

Если же сортирующее перемещение стартует со стержня 3, то всё совершенно аналогично, даже раписывать не буду. Если необходимо, то всё отчётливо видно в коде.

А вот и код: **пример 05.sorting.go**

```
package main

import (
    "fmt"
)
```

```go
func Hanoy(n int, start, transit, finish int) {
    if n > 0 {
        Hanoy(n-1, start, finish, transit)
        fmt.Println(start, finish)
        Hanoy(n-1, transit, start, finish)
    }
}

func Hanoy2(n int, start int) {
    if n > 0 {
        switch {
        case start == 1 && n%2 == 0:
            Hanoy(n-1, 1, 2, 3)
            fmt.Println(1, 2)
            Hanoy2(n-1, 3)
        case start == 1 && n%2 == 1:
            Hanoy(n-1, 1, 3, 2)
            fmt.Println(1, 3)
            Hanoy2(n-1, 2)
        case start == 2  && n % 2 == 0:
            Hanoy2(n-1, 2)
        case start == 2  && n % 2 == 1:
            Hanoy(n-1, 2, 3, 1)
            fmt.Println(2, 3)
            Hanoy2(n-1, 1)
        case start == 3  && n % 2 == 0:
            Hanoy(n-1, 3, 2, 1)
            fmt.Println(3, 2)
            Hanoy2(n-1, 1)
        case start == 3  && n % 2 == 1:
            Hanoy2(n-1, 3)
        }
    }
```

```
}

func main() {
    Hanoy2(8, 1)
}
```

Сортирующие башни. Обобщение.

Рассмотрим такой вариант: в начале все кольца лежат на первом стержне, но задаётся конечная позиция - какое кольцо на каком стержне должно находиться. Допустимо, чтобы некоторые кольца в конечной позиции могут находится и на первом стержне. Конечно, в конечной позиции большее кольцо не должно лежать на меньшем.
Интересный и не совсем тривиальный в техническом плане вариант для самостоятельной работы.

Ханойские башни. Несправедливые башни.

В классичесой задаче вводим дополнительное ограничение на список возможных ходов: запрещено класть кольцо 1 на стержень 2.

Решение.

И опять абсолютно тот же подход. Нам надо переложить кольцо N, для этого надо разместить всю пирамиду высоты N-1 на одном каком-то стержне. И этот стержень не есть стержень 2. Остаётся только стержень 3. Итак, перемещаем пирамиду высоты N-1 с первого стержня на третий несправедливым образом, после этого перекладываем кольцо N с первого стержня на второй. А дальше всё понятно: перемещаем всю пирамиду высоты N-1 с третьего стержня на первый (несправедливо,

конечно), перекладываем кольцо N со стержня 2 на стержень 3 и, наконец, снова перемещаем всю пирамиду высоты N-1 с первого стержня на третий. Это всё мы делаем, если N > 1. Если N = 1, то всё иначе и проще. Перемещать надо башню высоты 1, а это уже не башня, это просто одно кольцо 1. И его нельзя класть на стержень 2. Но это и не требуется - ведь в процессе перемещения более высоких башен кольцо 1 никогда не окажется на стержне 2. Но оно одно, это кольцо 1. Так что просто переносим его одним движением куда надо.

Код вот: **пример** 06.unfair.go

```
package main

import (
    "fmt"
)

func Hanoy(n int, start, finish int) {
    if n == 1 {
        fmt.Println(start, finish)
    }
    if n > 1 {
        Hanoy(n-1, start, finish)
        fmt.Println(start, 2)
        Hanoy(n-1, finish, start)
        fmt.Println(2, finish)
        Hanoy(n-1, start, finish)
    }
}

func main() {
    Hanoy(4, 1, 3)
}
```

Ханойские башни. Обменные башни.

Исходная и конечная позиция - классические. А вот набор возможных ходов другой. Кольцо 1 можно перекладывать как угодно. Кроме того, разрешено менять местами два кольца, лежащих на вершинах разных башен, если их размеры отличаются на 1.

Решение.

1. Переносим пирамиду высоты N-1 с одного стержня на другой, используя третий стержень как вспомогательный.
2. Переносим пирамиду высоты N-2 с другого стержня на третий.
3. Меняем местами кольца N и N-1.
4. Переносим пирамиду высоты N-2 со своего на кольцо N-1.
5. Переносим пирамиду высоты N-1 со своего на кольцо N.

Да, решение получается не оптимальным по количеству ходов. Есть, конечно, метод построить оптимальное решение (и доказать его оптимальность), но мы здесь это решение рассматривать не будем - уж очень это уведёт нас в сторону.

Иллюстрация решения - здесь

Код полностью соотвествует приведённому решению: **пример 07.unfair.go**

```
package main

import (
    "fmt"
)
```

```
func Hanoy(n int, start, transit, finish int) {
    if n == 1 {
        fmt.Println("  1:", start, finish)
    }
    if n > 1 {
        Hanoy(n-1, start, transit, finish)
        Hanoy(n-2, finish, start, transit)
        fmt.Println("swap", start, finish)
        Hanoy(n-2, transit, finish, start)
        Hanoy(n-1, start, transit, finish)
    }
}

func main() {
    Hanoy(4, 1, 2, 3)
}
```

… и другие головоломки.

Светофор

N ламп расположены в ряд. Каждая лампа может светить красным, жёлтым или зелёным цветом. Сначала они все светят красным. Задача состоит в том, чтобы все лампы светились зелёным. При этом разрешается переключать цвет ламп по следующим правилам:

- за один раз разрешается переключать только одну лампу;
- лампу можно переключить, если все лампы слева светятся одинаково, и их цвет отличается от цвета переключаемой лапы; новый цвет лампы также должен отличаться от цвета всех ламп,

стоящих левее.

Анализ задачи. Присмотревшись, мы увидим, что это точная копия классической задачи о Ханойских башнях: три стержня - это три цвета, N дисков — это N ламп, перемещение диска - это переключение цвета лампы, а размер диска - это номер позиции лампы, считая слева. Так что решение этой задачи есть точное повторени решения Ханойских башен, надо только изменить слова. Давайте на всякий случай сформулируем это решение:

- при N = 1 сразу выполнить необходимое перекрашивание
- при N > 1
  - рекурсивно переключить первые N − 1 ламп из начального цвета во вспомогательный
  - последнюю, N-ю лампу переключить из начального цвета в конечный
  - рекурсивно переключить первые N − 1 ламп из вспомогательного цвета в конечный.

Ещё один вариант. Если вполне естественно допустить переключение ламп из красного цвета только в жёлтый, и из зелёного тоже только в жёлтый (а из жёлтого, разумеется, в любой), то мы получим Ханойский Ремонт-2.

Да, ничего нового в решении нет. Но реализация, если визуализовать ход решения, потребует некоотрых усилий, так что можно использовать эту задачу для самостоятельной работы, для отработки технических навыков. Но с точки зрения рекурсии здесь, конечно, вообще нет ничего нового.

Переправа

Имеется клетчатая доска шириной в N клеток и высотой в 3 клетки.

Нижняя горизонталь заполнена фишками. Задача — передвинуть все фишки на верхнюю горизонталь. Оди ходом фишка ходит на любую клетку, но только по вертикали вверх или вниз. При этом запрещено

1. становиться на клетку, левее которой есть хотя бы одна фишка
2. двигать фишку, левее которой есть хотя бы одна фишка.

Анализ. И снова Ханойские башни в чистом виде. Три горизонтали - это три стержня. Первая слева фишка представляет наименьший диск и блокирует все правые клетки. В такой ситуации единственно возможный ход — это ход первой шашкой. Решение опять повторяется, только в других терминах.

И в этой задаче нет ничего нвого с точки зрения рекурсии. Но можно использовать и эту задачу для самостоятельной работы, для отработки технических навыков. Речь, конечно, идёт о визуализации процесса, не об алгоритме решения.

Ханойские лестницы, или Junk’s Hanoi

Junk’s Hanoi - это в честь изобретателя головоломки Junk Kato. Тут нужна картинка и для описания задачи и для описания логики решения. Поэтому я в этом файле ничего писать не буду, а отошлю к **файлу stairs.Task&Solution.pdf - к картинкам в pdf-формате**. Там приведены и условие задачи, и логика её решения.

Задача похожа по логике решения на Ханойские башни, но в это раз их решения всё-таки отличаются, это разные задачи. Программа должна выводить номера перемещаемых блоков и в какой угол их переводить. Строится программная реализация легко, так что приводить её не буду.

Лампочки. Опять.

Имеется ряд из N лампочек, пронумерованных от 1 до N слева направо. В начале все лампочки выключены. Требуется включить все лампочки, но, как водится, включая и выключая их по таким правилам:

- в любой момент можно переключить лампочку номер 1; переключить лампочку означает изменить её состояние: выключенную лампоку включить, включенную - выключить.
- разрешается переключить лапочку стоящую в соседней справа клетке от самой левой включенной лампочки.

Пример решения для N = 3. Записываем номера только горящих лампочек:

```
1 -> 1 2 -> 2 -> 2 3 -> 1 2 3 .
```

Логика решения.

Включаем ряд из N лампочек.

Если N > 1:

1. Рекурсивно включаем первые N-1 лампочек.
2. Рекурсивно выключаем первые N-2 лампочек. В результате остаётся гореть только лампочка N-1.
3. Включаем ламочку N. Теперь горят лампочки N-1 и N.
4. Рекурсивно включаем первые N-2 лампочек. Получаем ряд из N горящих лампочек.

Но возникает вопрос о выключении ряда лампочек. Решим его.

Чтобы выключить ряд из N лампочек (N > 1), необходимо

1. Рекурсивно выключаем первые N-2 лампочек.
2. Выключаем ламочку N. Остаётся гореть лампочка N-1.
3. Рекурсивно включаем первые N-2 лампочек.
4. Рекурсивно выключаем первые N-1 лампочек.

Да, а лампочку номер 1 мы можем включить и выключить в любой момент без всякой подготовки.

И у этой задачи логика решения во многом совпадает с логикой решения Ханойских башен, но всё-таки отличия заметные. И прежде всего тут появляется непрямая (косвенная) рекурсия - TurnOn вызывает TurnOff, а TurnOff вызывает TurnOn. Хотя тут есть и прямая рекурсия - и TurnOn, и TurnOff вызывают сами себя, но косвенная рекурсия тоже есть, и она тоже рекурсия, хоть и не такая очевидная, как прямая (A вызывает A). Программная реализация строится легко, но, на всякий случай, вот она - в файле bulbs.go :

```
package main

import (
    "fmt"
)

func TurnOff (n int) {
    if n > 1 {
        TurnOff(n-2)
        fmt.Println(n)
        TurnOn(n-2)
        TurnOff(n-1)
    } else
    if n == 1 {
        fmt.Println(1)
    }
}

func TurnOn (n int) {
    if n > 1 {
        TurnOn(n-1)
        TurnOff(n-2)
        fmt.Println(n)
```

```
        TurnOn(n-2)
    } else
    if n == 1 {
        fmt.Println(1)
    }
}

func main() {
    TurnOn(6)
}
```

Практика

Отдельные задачи на практику в этот раз вроде и ни к чему. Более чем достаточно задач уже приведено выше. Ясно, что все вышеперечисленные задачи разобрать на лекции невозможно, да и вряд ли нужно. Насколько глубоко разбирать задачи, которые мы реим дать на практику - по обстоятельствам.

Да, и ещё неплохо бы какую-нибудь задачу из жизни Ханойских башен визуализовать. Пусть в текстовом формате, не нужно тут грузиться графикой, достаточно termbox'а. Приятность тут в том, что башня - это стек. Так что это будет хорошее упражнение на связные списки. Если, конечно, реализовывать башни-стеки на связных списках. Так что пусть дети так их и реализовывают.
Можно ограничиться переходным вариантом - не визуализовать процесс перекладываний в Ханойских башнях, а только проверять корректность каждого хода. Это избавляет нас от технического геморроя, связанного с

визуализацией, но оставляет необходимость хранить и поддерживать состояние стержней - что и в каком порядке на каждом стержне находится. Так что хранить башни как стеки всё равно придётся, и хорошо бы реализовывать стек на связном списке.